Homework 1:

3. Schema

*Customer(customer-name, street, city)*

*Branch(branch-name, city)*

*Account(customer-name, branch-name, account-number)*

**(a) Find the names of all customers who have an account in the ‘Region12’ branch.**

πcustomer−name(σbranch−name=′Region12′ (Account))

SELECT DISTINCT customer-name

FROM Account

WHERE branch-name = 'Region12'

**(b) Find the names of all customers who have an account in a branch NOT located in the same city that they live in.**

πcustomer−name(σA.city<>B.city∧A.branch−name=B.branch−name(ρB(Branch)× ρA(Customer ◃▹ Account)))

(i) Implicit cross join:

SELECT DISTINCT A.customer-name

FROM Account A, Branch B, Customer C

WHERE A.customer-name = C.customer-name AND A.branch-name = B.branch-name AND B.city <> C.city

(ii) Explicit join:

SELECT DISTINCT A.customer-name

FROM Account A JOIN Branch B ON A.branch-name = B.branch-name JOIN Customer C ON A.customer-name = C.customer-name

WHERE B.city <> C.city

**(c) Find the branches that do not have any accounts.**

πbranch−name(Branch) − πbranch−name(Account)

(i) Not in:

SELECT DISTINCT branch-name

FROM Branch

WHERE branch-name NOT IN (SELECT branch-name FROM Account)

**(d) Find the customer names who do not have any account in the ‘Region12’ branch.**

πcustomer−name(Customer) − πcustomer−name(σbranch−name=′Region12′ (Account))

(i) Except:

SELECT customer-name

FROM Branch

EXCEPT

SELECT customer-name

FROM Account

WHERE branch-name = 'Region12'

**(e) Find the customer names who have accounts in all the branches located in ‘Los Angeles’.**

πcustomer−name(Customer)− πcustomer−name(πcustomer−name(Customer) × πbranch−name(σcity=′LosAngeles′ (Branch))− πcustomer−name,branch−name(Account)) ／／／ Customer ÷ (σcity=′LosAngeles ′ (Branch))

(ii) Not In: (i) Except:

SELECT DISTINCT customer-name SELECT customer-name

FROM Customer FROM Customer

WHERE customer-name NOT IN EXCEPT

(SELECT customer-name SELECT customer-name FROM Branch B, Customer C

FROM Branch B, Customer C WHERE B.city = 'Los Angeles' AND

WHERE B.city = 'Los Angeles' AND (C.customer-name, B.branch-name) NOT IN (C.customer-name, B.branch-name) NOT IN

(SELECT customer-name, branch-name (SELECT customer-name, branch-name

FROM Account) ) FROM Account)

(iii) Count:

SELECT customer-name

FROM Account AS A, Branch AS B

WHERE A.branch-name=B.branch-name AND B.city = 'Los Angeles'

GROUP BY customer-name

HAVING count(DISTINCT B.branch-name) =

(SELECT count(DISTINCT branch-name)

FROM Branch

WHERE city='Los Angeles')

**(f) Find the customer names who have only one account.**

πcustomer−name(Customer)− πA.customer−name (σA.branch−name<>B.branch−name∨A.account−number<>B.account−number)∧A.customer−name=B.customer−name (ρA(Account) × ρB(Account)))

(i) Except:

SELECT customer-name

FROM Customer

EXCEPT

SELECT A.customer-name

FROM Account A, Account B

WHERE (A.branch-name <> B.branch-name OR A.account-number <> B.account-number) AND A.customer-name = B.customer-name

(ii) Not In:

SELECT DISTINCT customer-name

FROM Customer

WHERE customer-name NOT IN (i)

(iii) Aggregate:

SELECT customer-name

FROM Account

GROUP BY customer-name

HAVING count(DISTINCT account-number)=1

4. Schema: *Student(sid, GPA)*

**Write a relational algebra that finds the ids of the students with the lowest GPA.**

πsid(Student) − πA.sid(σA.GP A>B.GP A∧A.sid<>B.sid(ρA(Student) × ρB(Student)))

(ii) Not In:

SELECT DISTINCT sid

FROM Student

WHERE sid NOT IN

(SELECT A.sid

FROM Student A, Student B

WHERE A.GPA > B.GPA AND A.sid <> B.sid)

==================================================================

Homework 2:

1. Schema

*Employee(person-name, age, street, city)*

*Work(person-name, company-name, salary)*

*Company(company-name, city)*

*Manage(person-name, manager-name)*

**(a) Write a query in SQL to find the names of persons who work in one or more companies where they make a salary that is less than $22,000.**

SELECT person-name FROM Work WHERE Work.salary

**(b) Write an SQL query to find the names of persons who work in one or more companies and make less than $22,000 in the majority (i.e., 50% or more) of the companies they work for.**

SELECT Total.person-name

FROM (SELECT person-name, count(\*) as cnt FROM Work GROUP BY person-name) as Total,

(SELECT person-name, count(\*)) as cnt FROM Work

WHERE Work.salary= 0.5 \* Total.cnt

**2. (a) Find the name(s) of the employee(s) whose total salary is higher than those of all employees living in Barstow.**

SELECT person-name

FROM Work

GROUP BY person-name

HAVING SUM(salary)>ALL

(SELECT SUM(salary) FROM Work, Employee

WHERE Work.person-name=Employee.person-name AND city=’Barstow’

GROUP BY Work.person-name)

======

SELECT person-name

FROM Employee E

WHERE NOT EXISTS

(SELECT Work.person-name FROM Work, Employee

WHERE Work.person-name=Employee.person-name AND city=’Barstow’

GROUP BY Work.person-name

HAVING SUM(salary)>=

(SELECT SUM(salary) FROM Work W

WHERE W.person-name=E.person-name))

**(b) Find the name(s) of the manager(s) whose total salary is higher than that of at least one employee that they manage**

SELECT manager-name

FROM Manage M,

(SELECT person-name, SUM(salary) total-salary FROM Work GROUP BY person-name) S1

WHERE M.manager-name=S1.person-name AND S1.total-salary > SOME

(SELECT total-salary

FROM (SELECT person-name, SUM(salary) total-salary FROM Work GROUP BY person-name) S2

WHERE S2.person-name = M.person-name)

======

SELECT manager-name FROM Manage M WHERE EXISTS (SELECT \* FROM (SELECT person-name, SUM(salary) total-salary FROM Work GROUP BY person-name) S1, (SELECT person-name, SUM(salary) total-salary FROM Work GROUP BY person-name) S2 WHERE M.manager-name=S1.person-name AND M.person-name = S2.person-name AND S1.total-salary > S2.total-salary)

3. Schema

*MovieStar(name, address, gender)*

*MovieExec(name, address, company, netWorth)*

**(a) We want to find the names and addresses of all female movie stars (gender = ’F’ in the MovieStar relation) who are also movie executives with a net worth over $2,000,000 (netWorth > 2000000 in the MovieExec relation).**

(i) Using INTERSECT: SELECT name, address FROM MovieStar WHERE gender=’F’ INTERSECT SELECT name, address FROM MovieExec WHERE netWorth>2000000

(ii) Without INTERSECT: SELECT name, address FROM MovieStar WHERE gender=’F’ AND (name, address) in (SELECT name, address FROM MovieExec WHERE netWorth>2000000)

**(b) We want to find the movie stars who are not movie executives.**

(i) Using EXCEPT: SELECT name FROM MovieStar EXCEPT SELECT name FROM MovieExec

(ii) Without EXCEPT: SELECT name FROM MovieStar WHERE name not in (SELECT name FROM MovieExec)

4. Schema: (NOTE: Computer product is either a desktop or laptop)

*ComputerProduct(manufacturer, model, price)*

*Desktop(model, speed, ram, hdd)*

*Laptop(model, speed, ram, hdd, weight)*

**(c) Find the average price of PC’s and laptops made by “Dell.”**

SELECT AVG(price) FROM ComputerProduct WHERE manufacturer=‘DELL’

**(e) Find the manufacturers that make at least three different computer models.**

SELECT manufacturer FROM ComputerProduct GROUP BY manufacturer HAVING COUNT(model)>=3

**5. (a) Using two INSERT statements, insert a desktop computer manufactured by HP, with model number 1200, price $1000, speed 1.2Ghz, 256MB RAM, and an 80GB hard drive.**

INSERT INTO ComputerProduct VALUES (‘HP’, 1200, 1000); INSERT INTO Desktop VALUES (1200, ‘1.2GHz’, ‘256MB’, ‘80GB’);

**(b) Using two DELETE statements, delete all desktops manufactured by IBM with price below $1000.**

DELETE FROM Desktop WHERE model IN (SELECT model FROM ComputerProduct WHERE manufacturer = ‘IBM’ AND price < 1000);

DELETE FROM ComputerProduct WHERE manufacturer=‘IBM’ AND price <1000 AND Model Not in (Select model from LapTop)

**(c) For each laptop made by Gateway, add one kilogram to the weight. (Hint: The WHERE clause in a**

UPDATE Laptop SET weight=weight+1

WHERE model IN (SELECT model FROM ComputerProduct WHERE manufacturer=‘Gateway’);

6. Schema: *Enroll(sid, dept, cnum, sec)*

**(a) Write an SQL query to find the students who are only enrolled in the CS classes offered this quarter.**

SELECT E1.sid

FROM Enroll AS E1

WHERE E1.sid NOT IN (SELECT Sid FROM Enroll WHERE dept<> 'CS')

**(b) Write an SQL query to find the students who are enrolled in all the CS classes offered this quarter.**

SELECT E0.sid /\* an enrolled student who is not missing any CS class \*/

FROM Enroll AS E0

WHERE E0.sid NOT IN /\* a E1.sid is a student who is missing some CS class \*/

(SELECT E1.sid FROM Enroll AS E1

WHERE (E1.Stid, E1.dept, E1.cnum)

NOT IN (SELECT E1.Stid, 'CS', E2.cnum

FROM Enroll AS E2 WHERE E2.dept='CS'))

**(c) Write the previous queries using different SQL constructs. In particular can you express those queries using the count aggregate? Please explain**

SELECT E1.sid

FROM Enroll AS E1

WHERE E1.dept ='CS'

GROUP BY E1.sid

HAVING count(\*) = (SELECT count(\*) FROM Enroll WHERE dept= 'CS')

**Show the view definition statements for EmployeeNames and DeptInfo.**

CREATE VIEW EmployeeNames AS

SELECT ename

FROM Employees;

CREATE VIEW DeptInfo AS

SELECT dept, AVG(Salary) avgs

FROM Employees

Group by dept;

Homework 3:

1. We want to store the table created by the following SQL statement into a disk.

CREATE TABLE Class(

dept CHAR(2), cnum INTEGER, sec INTEGER,

unit INTEGER, year INTEGER, quarter INTEGER,

title CHAR(30), instructor CHAR(20) )

We need to store tuples for 1,000 classes that have been offered so far. 10 classes are offered every year. The tuples are stored in random order (i.e., they are not sequenced by any attribute). A disk of the following parameters is used for storing the table.

• 3 platters (6 surfaces) • 10,000 cylinders

• 500 sectors per track • 1024 bytes per sector

• 6,000 RPM rotational speed • 10ms average seek time

**(a) What is the capacity of this disk?**

6 surfaces/disk \* 10,000 tracks/surface \* 500 sectors/track \* 1KB/sector = 30GB/disk

**(b) What is the average time to read a random sector from the disk?**

time for one rotation = 60000/6000 = 10ms rotation delay = 10ms\*1/2 = 5ms transfer time = (1/500)\*10ms = 0.02ms

seek time + rotation delay + transfer time = 10ms + 5ms + 0.02ms =15.02ms

**(c) Assume one disk block corresponds to one disk sector. How many disk blocks are needed to store the above table with 1,000 tuples?**

Floor(1024/72) = 14 tuples/block ceil[(1000 tuples/table)/(14 tuples/block)] = 72 blocks/table.

**(d) We want to run the following query by scanning the entire table.**

SELECT \* FROM Class WHERE year = 2005

Assuming that all blocks for the table is allocated sequentially, how long will it take to run the query? Assume that the disk head is not on the same track where the first block of the table is stored.

(seek time) + (rotational delay) + (transfer time) = 10ms + 5ms + 72\*0.02ms = 16.44ms

**(e) Now assume that due to frequent updates to the table, disk blocks are allocated such that, on average, sequentiality is broken every three blocks. That is, the table is stored in 24 randomly located “clusters” of 3 consecutive blocks. Assuming that we scan the entire table to execute the above query, how long will it take?**

24 \* ((seek time) + (rotational delay) + (transfer time)) = 24 \* (10ms + 5ms + 3\*0.02ms) = 361.44ms

**(f) Now assume that we have a B+tree on the year attribute and the tree has already been loaded into main memory. None of the disk blocks containing the Class table has been cached in main memory. What is the expected time to run the above query? Is it helpful to create a B+tree to run this query?**

150.2ms. Since the tuples are not clustered by the search key, we will need to do 10 random IOs to retrieve all 10 tuples. Therefore, 10\*(10ms+5ms+0.02ms) = 150.2ms. If all blocks are allocated sequentially, using the index may actually slow down the query execution.

2. Indexes:

The table taken(StNo, CourseID, Year, Quarter, Sec, Grade, Remarks) contains the grades for the courses completed by UCLA students during the last 20 years. If 10,000 new students enter UCLA every year, we can assume that in taken there are 200,000 different students, each identified by a StudentID. Thus will assume there are 40,000 students enrolled each quarter, and that each student takes four classes per quarter (160,000 classes taken by students each quarter) and that there are three quarters in each year (480,000 classes taken every year). Thus we get a total of 9,600,000 tuples recording the grades of all students over the last 20 years. Also assume that the average number of students per class is 100; this implies that 4,800 classes are o↵ered each year. On this table, we have a sparse index on StNo and a dense index on the combination: (CourseID,Year,Quarter,Sec,StNo). Both indexes are implemented as B+ trees where CourseNo, Year, Quarter, Sec, and StNo take 8 bytes each, and the pointers used in the B+ trees take 10 bytes.

**(a) If the file blocks have 4096 bytes and each tuple in taken requires 100 bytes, how many blocks will be needed to store the unspanned tuples of this relation ?**

floor(4096/100) = 40 unspanned tuples per block

ceil(9,600,000/40) = 240,000 blocks total

**(b) Compute the levels and the number of blocks at each level of the B+ tree, assuming a worst-case scenario**

Key: 8 bytes per field, with 5 fields = 40B.

block >= n\*ptr + (n – 1)\*key; 4096 >= 10n + 40n – 40, n = 82

N = floor[(4096-10)/50] + 1 = 82 (worst case: why floor not ceiling?)

N/2 = 41 pointers (worst case leaf and internal nodes)

Dense index: floor(9,600,000/41) = 234,146 at leaf level

floor(234,146/41) = 5,710 at first level

floor(5,710/41) = 139 at second level

floor(139/41) = 3 at third level (finally at root level)

**(c) How many blocks of B+ tree and file will the DBMS retrieve from disk to answer the following query: Find the average grade in a given class (e.g. find the average grade for: CS143, 2010, Fall, sec. 1). Assume the worst-case scenario, and that all the buffers are initially empty**

We can use the dense index because the search key is a prefix of the dense index key. 5 index blocks to the first leaf, then we need to retrieve leaf pointers for all 100 students. That’s up to three additional leaf blocks worst-case. Assuming that 100 students took that particular class, 100 more blocks will be accessed for actual data records, since this is not a a clustered index. Total: 8 B+ tree blocks, 100 file blocks.

**(d) We now want to compute the average grade over the (480,000 or so) classes taken in year 2011 (assume that they all have the same credit). Explain how the DBMS will go about searching and retrieving blocks from disk for this query, and estimate the number of blocks the system will have to fetch if those 200,000 students each took 48 classes on the average. Assume the worst-case scenario, and that all the buffers are initially empty.**

We will have to scan the whole file with the help of the sparse, and thus clustered, index on StNo.

floor[(4096-10)/18] = 227 (thus N for B+ tree is 228)

114 w.c. pointers. So, with 240,000 blocks, we see floor(240,000/114) = 2,105 blocks at leaf level. So there are 18 blocks at the next level, and then the root. The complete file is fetched by following the leftmost branch in the B+ tree and then the leaf nodes that are chained together. Thus we have to fetch 2 + 2,105 + 240,000 blocks.

3. Joins and Optimization:

In addition to the table taken whose schema and index have been described previously, we also have the table student(StNo,Level,FirstName,LastName,Major) describing our 200,000 students. This table has a primary B+ index on StNo which is the key for this relation and a foreign key for taken.There are five di↵erent levels: freshman, softmore, junior, senior, others

**(a) How many blocks will student use, if each tuple requires 100 bytes and each block contains 4096 bytes?**

200,000/40=5,000 blocks

**(b) For the query πStNo(σ=“others”(student)) ▹◃ taken estimate the size of the results (i.e., how many tuples). Also estimate the cost of implementing the query measured by the number if blocks read from disks. You can assume that the join takes advantage of the sparse index on taken.StNo (Also, for the sake of simplicity, assume that all indexes used are already in main memory). To compute these estimates use the textbook rules of relational queryoptimizers.**

Because of the FK the complete join has as many tuples as taken 9, 600,000 tuples. Others is one out of five, 9, 600,000/5= 1920,000. Computing the join using the index. 200,000/5=40,000 ”others” students. The courses taken by one student fit in two blocks. Thus we have to access 40000 ⇥ 2 blocks of taken plus the 5000 blocks of student

==================================================================

Sample Midterm:

1. Suppose that blocks can hold 100 search keys and 101 pointers. We built a dense index organized as a B+ tree on a file of 2 million records, where the records are placed in blocks that hold 10 records each. Say that the search key for the B+ tree is the candidate key for the relation. Answer in the worst-case scenario:

**(a) Compute the blocks used at each level of the B+ tree**

Candidate key: as many pointers as there are records (2 million pointers)

Bottom level: 2 million/50 = 40,000

Next level: floor(40,000/51) = 784 pointers

Next level: 784/51 = 15 blocks. Then the root

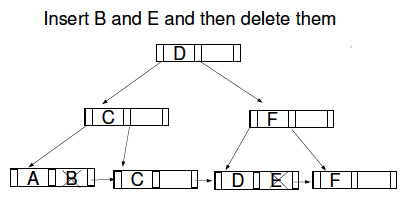
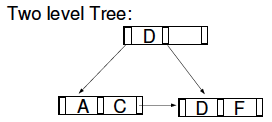
Total number of blocks = 40000 + 784 + 15 + 1

(i) If this was a sparse index, we only need 200,000 pointers. At bottom level = 4,000. Next level = 78. Then root

2. Assume that we use B+ trees of order n = 3 (where n is the max number of pointers in a node)

**(i) Draw a tree of height 2 with the following keys: A, C, D, F (lexicographically ordered)**

**(ii) Show how the insertion of two new keys, followed by their deletions, could change this 2-level tree into a 3-level tree with exactly the same keys (i.e. A, C, D, F)**



3. We store the following file in blocks having size 4096 bytes. Relation: *customer(id char(20), name char(24))*

Our relation has 2 millions of tuples, which are stored unspanned. We create sparse index on id organized as a B+ tree. Pointers take 10 bytes.

**(a) How many blocks are needed to store the whole relation?**

Since 4096/44 = 93.09 (we can store 93 unspanned tuples in each block)

Thus, ceil(2,000,000/93) = 21,506 blocks

**(b) What is the minimum number of nodes needed for the B+ tree?**

N – 1 = ceil[(4096-10)/(20+10)] = 136. Thus N = 137

Leaf Level: 137 – 1 = 136 pointers to the file. Sparse index pointing at 21,506 blocks. Since 21,506/136 = 158.13, 159 leaf nodes are used

First Level: N = 137 pointers used. Thus, ceil(159/137) = 2 blocks used

1 Root: Thus the total number of nodes in the best case is 1 + 2 + 159 = 162

**(c) What is the maximum number of nodes needed for the B+ tree, in the worst-case scenario?**

N is still 137

Leaf Level: ceil[(137+1)/2) = 69. Thus, 68 pointers to the file in each leaf node. Sparse index pointing to 21,506 blocks. Since 21,506/68 = 316.2, we conclude that 316 blocks are used at leaf level

First level: ceil(137/2) = 69 pointers in each node. Thus, floor(316/69) = 4 nodes needed

1 Root: Thus total nodes: 1 + 4 + 316 = 321

**(d) Using the worst-case B+ tree you just constructed, how many blocks must be retrieved**

**to execute the following query:**

**SELECT name FROM customer WHERE id = '7672945'**

3 blocks of the B+ tree and one from the file. Four all together.

3. Given the table taken(StNo, CourseID, Year, Quarter, Sec, Grade, Remarks):

**(a) Write a SQL query to find the students who have taken 4 or more classes and, in every class they took, got a grade that is equal to or lower than the average for that class—a class is identified by (CourseID, Year, Quarter, Sec) and Grade in taken is of type numeric.**

We are seeking students who never got a grade above the class average and took four or more classes:

select StNO from taken where

StNO not in ( select StNo

from taken as T1

where Grade > (select avg(Grade) from taken as T2 where T2.CourseID=T1.CourseID and T2.Year= T1.Year and T1.Quarter= T2.Quarter and T2.Sec= T1.Sec))

group by SNO having count(\*) >= 4

5. Given the table taken(StNo, CourseID, Year, Quarter, Sec, Grade, Remarks):

**(a) write a relational algebra expression to compute all the students who got a grade above 3.0 in every class they took in 2014.**

πStNo(taken) − πStNo(σGrade<=3^Year=2014(taken))

6. Potpourri

**(a) Can the intersection of relations R(A,B) and S(A,B) be expressed using only natural joins?** Yes, R ^ S = R >< S.

**(b) Can the intersection of relations R(A,B) and S(A,B) be expressed using the set difference operator?** Yes, R ^ S = R − (R − S).

**(c) Can the intersection of relations R(A,B) and S(A,B) be expressed using the Cartesian product and projection operators?** No

**(d) Can the intersection of relations R(A,B) and S(A,B) be expressed using the cartesian product, selection and projection operators?**

Yes, R ^ S = πR.A,R.B(σR.A=S.A^R.B=S.B(R x S)).

**(e) A relation R is indexed on its candidate key using an extendible hashing: Is this index dense, sparse or it could be either way?**

This must be a dense index. Sparse indexes only work if we can perform ordered searches as in B+ trees.

**(f) For the extendible hashing on R described in D5: does the structure of its directory and the number of the buckets it uses depend on the order in which the tuples in R have been inserted, or they only depend on the values those tuples?**

The overall structure depends only on the set of values. Different insertion orders might affect the order in which tuples are arranged in the buckets, but not the overall content of the buckets, nor the directory.

**7 (exc). Our RDBMS has compiled and optimized our relational query into a select-project-join expression consisting of 3 selections, 3 projections and 4 joins. Assume that the relations in our database are in main memory and each contains N tuples or less. Is the worst-case complexity of executing this query log(N), polynomial in N or exponential in N? Justify your answer**

The least efficient kind of join is the nested loop join that computes the four joins in time ((N x N) x N) x N = N^4. The resulting size is also less than N4 and selections are performed in linear time whereas projections by sorting are performed in time N4 x log(N^4), which is better than O(N^5). Thus in the worst case, the complexity is polynomial, and because of indexes and optimizer it will actually be better than O(N^5)

**In the space below, write a Relational Algebra expression that returns the sid of all students who have taken both CS162 and CS186 (where CS162 and CS186 are “cid”s) but no other courses. Do not use any unnecessary relations.**

****

SELECT DISTINCT profname FROM Courses =

SELECT profname FROM Courses GROUP BY profname =

SELECT profname FROM Courses UNION SELECT profname FROM Courses;

**When would the following two queries return different results for a given database instance?**

**Left outer join vs s.sid = r.sid**

When a student is not registered for course.

**Write a query in SQL to find the names of such companies that all of their employees have salaries higher than $100,000**

SELECT company-name

FROM Company C

WHERE 100000 < ALL

(SELECT salary FROM Work W

WHERE C.company-name = W.company-name)

**Find the name(s) of the employee(s) whose total salary is higher than those of all employees living in Los Angeles**

SELECT person-name FROM Work

GROUP BY person-name

HAVING SUM(salary) > ALL

(SELECT SUM(salary) FROM Work, Employee

WHERE Work.person-name = Employee.person-name AND city=’Los Angeles’

GROUP BY Work.person-name)

**Find the name(s) of the manager(s) whose total salary is higher than that of at least one employee that they manage.**

Total\_Salary:

SELECT person-name, SUM(salary) total-salary

FROM Work

GROUP BY person-name

SELECT manager-name

FROM Manage M

WHERE EXISTS

(SELECT \* FROM

Total\_Salary S1,

Total\_Salary S2

WHERE M.manager-name=S1.person-name AND

M.person-name = S2.person-name AND

S1.total-salary > S2.total-salary)

**Common Constraints**

• NOT NULL - Ensures that a column cannot have a NULL value

• UNIQUE - Ensures that all values in a column are different

• PRIMARY KEY - A combination of a NOT NULL and UNIQUE. Uniquely identifies each

row in a table

• FOREIGN KEY - Uniquely identifies a row/record in another table

• CHECK - Ensures that all values in a column satisfies a specific condition

• DEFAULT - Sets a default value for a column when no value is specified

• INDEX - Used to create and retrieve data from the database very quickly

B+ Trees:

- At least half of the pointers used (i.e. ceil(n/2))

- Except root, where at least 2 pointers used

- Non-leaf node: - Leaf node (always at least half full):

- Max pointers (n), min pointers (ceil(n/2)) - Max pointers (n), min pointers (ceil(n+1)/2)

- Max keys (n – 1), min keys ((ceil(n/2) – 1) - Max keys (n – 1), min keys (ceil((n-1)/2))

- Root: - Max pointers (n), min pointers (2) - Max keys (n – 1), min keys (1)

Insertion:

- There are four cases to consider:

1. Simple case (no overflow) 2. Leaf overflow only 3. Non-leaf overflow 4. New root

- Leaf overflow:

1. Traverse down to the correct leaf node, but there is no space to store the new value

2. “Leaf node splitting” => split the leaf into 2, and put the keys half and half (NOTE: Maintain search-key order)

3. Copy the first key of the new node (i.e. 60) to the parent node a. If there is no overflow in the parent node, then stop (otherwise, case 3)

- Non-leaf node overflow:

- Ex: Leaf node to insert (50, 55), parent node of that node (50, 60), and parent node to that parent node (70, null), and node to insert is 52

1. Traverse down to the leaf node (i.e. 50, 55)

2. Leaf node overflow, so split it, copy the key into the new node: left leaf node (50, 52), and right leaf node (55, null)

3. Copy the first key in the new node (i.e. 55) up the parent node: non-leaf node overflow (50, 55, 60)

4. Parent leaf overflow, so split it, and **move up the key in the middle**: Left parent node (50, null), right parent node (60, null)

a. Middle key (i.e. 55), moves up to that node’s parent node (i.e. (70, null)): so now (55, 70) (no overflow, so done)

- New root:

- Ex: Leaf node to insert into (20, 30), parent node/root node (50, 60), and key to insert is 25

1. Traverse to leaf node (i.e. (20, 30)), and insert 25 there, leading to node overflow: (20, 25, 30)

2. Split leaf node: left leaf node (20, 25), right left node (30, null)

3. Copy first key in the new node (i.e. 30) up to the parent node, leading to parent node overflow (30, 50, 60)

4. Split the parent node: left parent node (30, null), right parent node (60, null), and move up the middle key (i.e. 50)

5. As there is no node above, add new node, and place 50 in that node (i.e new root node)

- Summary:

- Leaf node overflow => the first key of the new node is copied to the parent

- Non-leaf node overflow => the middle key is moved up to the parent

====

Deletion:

- 5 Cases to consider:

2. Leaf node, coalesce with neighbor

3. Leaf node, redistribute with neighbor

4. Non-leaf node, coalesce with neighbor

5. Non-leaf node, redistribute with neighbor

- Leaf node, coalesce with neighbor:

- Ex: parent node a(20, 40, 60), children nodes b, c(20, 30, null), d(40, 50, null), e, and we want to delete 50

1. Delete 50, and check number of pointers => now we have 2 pointers (3 min), so underflow

2. Try merging with a sibling, merge c and d (i.e. move everything from d into c) => c(20, 30, 40), and update pointers (i.e. cn pointer points to e now, and not d)

3. Once everything is moved, delete d

4. After leaf node merge, from the parent node, delete the pointer and key to the deleted node (i.e. delete 40), and check for underflow at a

- Leaf node, redistribute with neighbor:

- Ex: parent node a(20, 40, 60), children nodes b, c(20, 25, 30), d(40, 50, null), e(60, 70, 80), and we want to delete 50

1. Delete 50, and check for underflow => 2 pointers (min 3), so underflow

2. Check if d can be merged with siblings c or e, and if not, then redistribute the keys in d with a sibling

a. Say c (for example) => so redistribute c and d, so that nodes c and d are roughly “half full”

- Move 30 and its tuple pointer to d (i.e. c(20, 25, null) and d(30, 40, null))

3. Update the parent node (as the lowest value in d is now 30 (and not 40), replace 40 with no in the parent node) => parent node a(20, 30, 60)

a. Check for underflow => 4 pointers (min 3), so done

- Non-leaf node, coalesce with neighbor:

- Ex: a(50, 90, null), a’s children are b(30, null, null) and c(70, null, null), b’s children are d(10, 20, null), and e(30, 40, null), c’s children are f(50, 60, null), and g(70, 80, 90), and we want to delete 20

1. Delete 20, and check for underflow => d(10, null, null) => 2 pointers (min 3) => so underflow

a. Merge d with e (if you can) => you can, so move everything in e to d => e is empty, d(10, 30, 40)

2. From parent node, delete pointer and key to the deleted node (i.e. delete 30) => check for underflow => 2 pointers (min 2) => so underflow

a. Try to merge with its sibling => b(null, null, null) and c(70, null, null)

- IMPORTANT NOTE: When merging non-leaf nodes => always pull down the mid-key in the parent and place it in the merged node

- So, move pointers from c to b, pull down 50, and bring over 70 from c => a(90, null, null), b(50, 70, null), and delete c

- Non-leaf node, redistribute with neighbor:

- Ex: a(50, 99, null), a’s children are b(30, null, null) and c(70, 90, 97), b’s children are d(10, 20, null) and e(30, 40, null), and c’s children are f(50, 60, null), and g(70, 100, 110), and we want to delete 20

1. Delete 20, and check for underflow => d(10, null, null) => underflow

2. Merge d with e (i.e. bring stuff in e to d) => d(10, 30, 40)

a. Remove key and pointer to deleted node from parent (i.e. remove 30) => b(null, null, null)

3. Try to merge b with c => b has 1 pointer, c has 4 pointers = 5 pointers (max 4) => can’t merge, so try to redistribute

a. Redistribute b and c:

i. Temporarily, make left node b “overflow” by pulling down mid-key and moving everything from c to b => b(50, 70, 90, 97), c(null,…), and a(99, null, null)

ii. Apply overflow handling algorithm:

a. pick the mid-key (say 90) in the node and move it to the parent => b(50, 70, 97) and a(90, 99, null)

b. Move everything to the right of 90 to the empty node => b(50, 70, null) and c(97, null, null)

- Summary of deleting:

- For leaf node merging => delete the mid-key from the parent

- For non-leaf node merging/redistributing => pull down mid-key from their parent

- In practice, coalescing is often not implemented (as it is too difficult and not worth it)

===============================

**- Create Trigger**

****

**Revoke grant option … cascade: 🡪 B Ans: B: Y,N C: Y,N D:N,N**

**Nested Loop Join:** bR+|R|\*bS = 100block + 1000tuple\*100block

**Block Loop Join:** br + br\*bs

**Bulk Block Loop Join:** bR+⌈bR/(M-2)⌉\*bS M: memory

Number of records of student: 5,000 takes: 10,000 ;; Number of blocks of student: 100 takes: 400

**Index Join :** 100 + 5000 \* 5 = 25,100 block transfers

**block nested loops join** : 400\*100 + 100 = 40,100 block

Suppose􀀁you􀀁have􀀁2􀀁relations,􀀁R(A,􀀁B)􀀁and􀀁S(B,􀀁C),􀀁with􀀁the􀀁following􀀁characteristics:

•􀀁 The􀀁size􀀁of􀀁one􀀁disk􀀁block􀀁is􀀁1000􀀁bytes.

•􀀁Attributes􀀁A,􀀁B􀀁are􀀁of􀀁length􀀁10􀀁bytes.􀀁Attribute􀀁C􀀁is􀀁of􀀁length􀀁180􀀁bytes.􀀁􀀼􀀖􀀑􀀳􀁍􀁐􀁄􀁌􀀾

•􀀁 The􀀁tuples􀀁are􀀁not􀀁spanned􀀁across􀀁disk􀀁blocks􀀁􀀁􀀁􀀼􀀖􀀁􀁕􀁖􀁑􀁍􀁆􀁔􀀁􀁑􀁆􀁓􀀁􀁃􀁍􀁐􀁄􀁌􀀾

•􀀁 |R|􀀁=􀀁5,􀀁000􀀁(number􀀁of􀀁tuples􀀁of􀀁R)􀀁􀀁􀀁􀀁􀀼􀀒􀀑􀀑􀀁􀀁􀁃􀁍􀁐􀁄􀁌􀁔􀀾

•􀀁|S|􀀁=􀀁500􀀁(number􀀁of􀀁tuples􀀁of􀀁S)􀀁􀀁􀀁􀀼􀀣􀀌􀀤􀀞􀀁􀀚􀀑􀀛􀀁􀀖􀀁􀀁􀀴􀀁􀁕􀁖􀁑􀁍􀁆􀁄􀁌􀁔􀀾

•􀀁We􀀁have􀀁30􀀁blocks􀀁of􀀁memory􀀁bu↵er

•􀀁 We􀀁use􀀁one􀀁disk􀀁block􀀁for􀀁one􀀁B+tree􀀁node

•􀀁 Each􀀁 pointer􀀁 in􀀁 a􀀁B+tree􀀁 index􀀁 (both􀀁 a􀀁 record􀀁 pointer􀀁 and􀀁 a􀀁 node􀀁 pointer)􀀁 uses􀀁 10

bytes.

**What is the minimum number of blocks needed to store R and S?**

R: 100(50/block) S: 100 (5/block)

**We want to construct a dense B+tree on attribute B of table S by scanning the S table. What is the maximum possible n for the B+tree given the above parameters?**

10n + 10(n-1)<1000 n = 50

**How many disk IOs would be incurred during the construction of the B+tree on S.B? Assume that you use the main memory buffer in the most efficient way to minimize the number of disk IOs.**

I/O cost: # blocks

Block of S +write back the B+ tree blocks. 121 = 100 +21

**Write a query to find the StID of all students who completed at least three classes with a grade B- or better. (The code for grades is numeric and the grade for B- is 2.7.**

Tk1 := Tk0 := ΠStID,CourseID(σGrade≥2.7(Taken))

Tk2 := ΠTk1.StID,Tk1.CourseID(Tk1 ⑅ Tk1.StID=Tk0.Stid ∧ Tk1.CourseID>Tk0.CourseID Tk0)

Result:= ΠTk2.StID(Tk2 ⑅ Tk1.StID=Tk2.Stid∧Tk2.CourseID>Tk1.CourseID Tk1)

**We aim to find suppliers who are the only suppliers of some part - in other words, they provide at least one part that no other supplier provides. One approach was to execute a self-join on the part number, ensuring a different supplier**.

SELECT SupplierNo FROM warehouse AS w1

WHERE NOT EXISTS (SELECT \* -- a second SupplierNo for w1.PartNo FROM warehouse w2 WHERE w1.PartNo = w2.PartNo AND w1.SupplierNo <> w2.SupplierNo)

**We aim to identify suppliers that sell at least two parts at the minimum price (for those parts)**

SELECT SupplierNo FROM warehouse w1

WHERE w1.Price = (SELECT MIN(Price) FROM warehouse w2 WHERE w1.partNo = w2.partNo)

GROUP BY w1.SupplierNo HAVING COUNT(\*) >= 2

**A2: The strict minimum check can be accomplished a few ways. One is to ensure that only one tuple exists for the given part number and price, eg adding the following to the WHERE clause of the previous A1(ii) solutions:**

[SELECT … FROM warehouse w1 WHERE … AND]

1 = (SELECT COUNT(\*) FROM warehouse AS w2 WHERE w1.PartNo = w2.PartNo AND w1.Price = w2.Price)

A3: DELETE FROM warehouse WHERE SupplierNo IN (query from A2)

A4: yes expressible. First we find parts that are supplied by more than one supplier:

../Desktop/Screen%20Shot%202020-02-11%20at%2014.04.09.png only one supplier:

../Desktop/Screen%20Shot%202020-02-11%20at%2014.04.14.png we can join against the warehouse table to find the corresponding supplier: ../Desktop/Screen%20Shot%202020-02-11%20at%2014.04.18.png First, we find all supplier+part combinations that cannot be the strict minimum. This means all supplier+part combinations where another supplier exists for the same part at cheaper or equal price: